2014 47th Annual IEEE/ACM International Symposium on Microarchitecture

Arbitrary Modulus Indexing

Jeffrey R. Diamond* Donald S. Fussell* Stephen W. Keckler*†
*The University of Texas at Austin †NVIDIA
{*jdiamond, fussell, skeckler*}*@cs.utexas.edu*

***Abstract*—Modern high performance processors require memory systems that can provide access to data at a rate that is well matched to the processor's computation rate. Common to such systems is the organization of memory into local high speed memory banks that can be accessed in parallel. Associative look up of values is made efficient through indexing instead of associative memories. These techniques lose effectiveness when data locations are not mapped uniformly to the banks or cache locations, leading to bottlenecks that arise from excess demand on a subset of locations. Address mapping is most easily performed by indexing the banks using a $\mathrm{mod}(2^N)$ indexing scheme, but such schemes interact poorly with the memory access patterns of many computations, making resource conflicts a significant memory system bottleneck. Previous work has assumed that prime moduli are the best choices to alleviate conflicts and has concentrated on finding efficient implementations for them. In this paper, we introduce a new scheme called Arbitrary Modulus Indexing (AMI) that can be implemented efficiently for all moduli, matching or improving the efficiency of the best existing schemes for primes while allowing great flexibility in choosing a modulus to optimize cost/performance trade-offs. We also demonstrate that, for a memory-intensive workload on a modern replay-style GPU architecture, prime moduli are not in general the best choices for memory bank and cache set mappings. Applying AMI to set of memory intensive benchmarks eliminates 98% of bank and set conflicts, resulting in an average speedup of 24% over an aggressive baseline system and a 64% average reduction in memory system replays at reasonable implementation cost.**

***Keywords*-prime banking; index schemes; fast division and modulus; GPU caches; replay architectures**

I. Introduction

Modern high-performance processors require memory systems that can provide access to data at a rate that is well matched to the computation rate. Common to such systems is the organization of memory into local high speed memory banks that can be accessed in parallel. Associative look up of values is made efficient through indexing instead of associative memories. These techniques lose effectiveness when data locations are not mapped uniformly to the banks or cache locations, leading to bottlenecks that arise from excess demand on a subset of locations. Address mapping is most easily performed by indexing the banks using some number N of low order address bits, i.e. by using a $\mathrm{mod}(2^N)$ indexing scheme. However, such schemes are poorly matched to the memory access patterns of many computations, making resource conflicts a significant memory system bottleneck.

Programmers can alleviate such interactions through various means, but these require significant effort, are specialized to specific situations, and are hard to maintain. Hardware solutions include various forms of bit hash indexing, which change but do not eliminate the most common conflicts due to power-of-2 divisors in the index and the memory access patterns. Indexing with non-power-of-2 moduli can alleviate many of these conflicts, but implementing such schemes efficiently in hardware is difficult. Work in this area has commonly assumed that prime moduli are the best choices to eliminate conflicts since they have the fewest divisors and has thus concentrated on efficient implementations for prime moduli.

The most efficient existing implementations are for moduli of the form $2^N - 1$, some of which are prime (Mersenne primes). For practical systems, only a few possible such choices exist, and they may not be of suitable size. This has led to efforts to find efficient implementations for other primes, including those of the form $2^N + 1$ [1]. While providing more flexibility, prime numbers that are not close to a power of 2 can be difficult to integrate into a power-of-2 based architecture, and indexing implementations for general moduli are relatively expensive. Approaches using prime moduli provide limited flexibility in the choice of modulus, especially in light of integration considerations, and the implementations have not yet proven to be attractive in practice.

In this paper, we introduce a new scheme called Arbitrary Modulus Indexing (AMI) that can be implemented efficiently for all moduli. AMI matches or improves the efficiency of the best existing schemes for non-power-of-2 indexing while allowing great flexibility in choosing a modulus to optimize cost/performance trade-offs. Our novel implementation of AMI requires less than 3% of the area of a 32-bit integer multiply unit, less than 0.5% of its power, and just a few gate delays.

As a case study, we evaluate the potential benefits of AMI when applied to a high throughput GPU memory system. Delivering high-bandwidth parallel access to memories requires heavily banking the RAM structures throughout the memory hierarchy. As a result, memory systems of GPUs are at least as reliant on efficient banking techniques as vector supercomputers. In addition, while many parallel algorithms on GPUs benefit from caching, the large numbers of threads put severe pressure on the on-chip caching structures including cache conflict misses. We address two major types

1072-4451/14 $31.00 © 2014 IEEE
DOI 10.1109/MICRO.2014.13

of conflicts in the primary memory system of a modern GPU: (1) *bank conflicts* that arise when multiple threads want to access the same level-1 cache or scratchpad memory bank to obtain different data in the same cycle; and (2) *set conflicts* due to the limited associativity in the cache banks. AMI enables the use of an arbitrary number of cache or scratchpad banks, which reduces many common cross-thread conflict patterns. Given the flexibility of AMI, we examine the performance of a set of memory-intensive benchmarks using a variety of moduli. Surprisingly, we find that the most promising Mersenne prime modulus (31) is not a good choice, and that some of the best moduli are not prime or even odd numbers.

Our results show that minimal additions to the memory system architecture reduce bank conflicts by over 98%, completely eliminating conflicts in 4 of the 5 benchmarks with the highest memory intensity. We also demonstrate that applying AMI to scratchpad banks, L1 cache banks, and L1 sets eliminates 64% of instruction replays, recovering essentially all of the performance lost from conflicts. Most importantly, we show that AMI offers tremendous design flexibility that enables several optimization trade-offs. The area and power overheads are more than offset by gains in performance and reduction in replays. Further, the additional latency required for arbitrary modulus computation is easily hidden in a latency tolerant GPU architecture.

The rest of the paper is organized as follows. Section II describes previous work on non-power-of-2 indexing schemes. Section III describes the AMI scheme and its implementation and compares it to existing efficient non-power-of-2 schemes. Section IV provides background on the architecture of throughput processors, discusses where AMI is applied, and details the architectural model used in this paper. Section V describes our simulation methodology, our power model and our choice of benchmarks for the study. Section VI characterizes the behavior of the throughput benchmarks used in this paper, particularly in terms of their conflict behavior, and shows how their behavior varies with choices of index moduli. Section VII quantifies the reduction in conflicts and improvements in performance stemming from AMI. Section VIII discusses conclusions and future work.

II. Related Work

As computers employ binary numbering schemes, hardware memory resources such as number of banks or sets are typically found in powers of 2. Likewise, software data structures are often accessed in a power-of-2 (or multiple thereof) stride as programmers optimize their algorithms for hardware implementations. Many have observed that mapping conflicts are worst when the index modulus and the memory access stride share a common divisor [2], [3], [4], so power-of-2 strides combined with power-of-2 address mapping schemes often lead to undesirable levels of resource conflicts. This problem has been well-studied in the context of interleaved memory bank and cache set conflicts, and both software and hardware approaches have been used to mitigate the conflicts.

A. Software Approaches

A programmer or compiler can carefully adjust the data layout of the code by padding array sizes or providing cyclic rotations of rows. Both library APIs [5] and language extensions [6] have been proposed to abstract array access and allow a programmer to independently specify the static memory layout of arrays. All of these approaches impose a burden on the the programmer and are impossible in cases where the size or access patterns of the data are dynamic [7]. While hardware solutions have been proposed to ease efficient data layout [8], it is preferable to reduce conflicts without modifying the structure of the data.

B. Power-of-2 Indexing Algorithms

Bit hashing: While numerous index hashing techniques that take the form of simple bit operations have been proposed [9], such schemes do not prevent all power-of-2 self-conflicts and thus end up trading off improved performance on some access patterns with reduced performance on others [10], [11]. Application-specific hashing schemes devised offline [12] or dynamically [13] reduce conflict misses by up to 30%, but AMI removes nearly all conflicts, even with a single, static indexing scheme across the benchmarks we examined.

Reordering: Buffering and reordering strided memory accesses has been shown to reduce bank conflicts [14], [15] within single streams of access. Reordering can be combined with advanced hash techniques such as Galois Fields [16] to statistically reduce conflicts for a general mix of vectors. However, SIMD lanes do not have per lane buffering or reordering, and conflict rates as low as 3% can halve throughput.

Associativity: Adding associativity to a direct-mapped cache can reduce bank conflicts but is costly to implement past a small number of ways. More sophisticated approaches use multiple bit-hash schemes, either on multiple ways, such as skewed-associative caches [9], or on the same way, such as column-associative caches [17], hash-rehash caches [18], ZCaches [19], [20], or various indirect indexing schemes [21], [8]. Many of these schemes are difficult to implement in practice. Approaches that increase associativity are primarily limited to reducing set conflicts in caches, as opposed to banks or scratchpads. In most cases, such approaches are orthogonal and complementary to non-power-of-2 caches, but have limited applicability to banks, scratchpads, small L1 caches, or modern GPUs. In addition, the parallel lookups required in associative caches increase power consumption; AMI can eliminate set conflicts without resorting to an associative cache.

C. Non-power-of-2 Indexing Algorithms

Prime number indexing: Prime moduli have long been assumed to be the best indexing choices to minimize conflicts since they have the fewest divisors. In some cases, prime bank indexing has proven to have sufficient advantages over simple power-of-2 mapping as to be worth implementing even as an expensive series of iterations [22].

As the latency and area required for index computation have become more critical, more efficient techniques for prime index computation have been sought [23]. The most significant optimizations in the prime index computation of general moduli involve the application of modulo/remainder algebraic properties, such as the Chinese Remainder Theorem (CRT) [24]. These approaches break an address into smaller chunks and apply a modulus operation by a linear combination of narrow address digits and constant weighting parameters. In the best case, when applying CRT to moduli of the form $2^N - 1$, the coefficients become 1 and the operation reduces to computing the modulus of a narrow sum of numbers [25].

Mersenne prime indexing: Given the relative efficiency of implementing moduli of the form $2^N - 1$ and the assumption that prime moduli are preferred [1], Mersenne primes of the form $2^N - 1$ have been the most efficiently implementable non-power-of-2 moduli studied. They also have the advantage over other primes of being close to powers of two and thus more easily integrated into hardware in which resources are sized in powers of 2 [10], [11]. However, Mersenne primes give few candidate moduli (the first 5 Mersenne primes are 3, 7, 31, 127 and 8191) and these are spread so far apart that there is often no appropriately sized choice. On the other hand, non-Mersenne primes are more plentiful but farther from powers of 2 and thus harder to integrate into other hardware. No sufficiently efficient implementations of non-primes other than powers of 2 and those of the form $2^N - 1$ [25] and $2^N + 1$ [1] have been demonstrated, perhaps because they have been assumed to be likely to underperform prime moduli. Thus finding a non-power-of-2 modulus that is effective at minimizing conflicts and efficiently implementable for a given application may be difficult.

III. Arbitrary Modulus Indexing (AMI)

We propose a flexible new approach to efficiently computing indices called Arbitrary Modulus Indexing (AMI). AMI can be implemented more efficiently than any previous non-power-of-2 indexing scheme. Our approach works not just for primes but for any positive integer, albeit with different area/delay costs depending on the exact number used. For about a third of the cases in our case study, AMI adds only a few gate delays to conventional base + index address generation, with area and power equivalent to a few narrow adds. Due to the generality of the approach, it is possible to use a common circuit that can be configured for a variety of moduli, making possible a dynamically configurable indexing modulus for cases where different workloads benefit from different moduli.

AMI has a novel derivation that stems from the original, unmodified hardware implementation of binary reciprocal array multiplication. We then optimize the algorithm and logic design for use in index computation. This approach guarantees an efficient hardware implementation and provides a clear view of how a given circuit implementation would support multiple choices of modulus. Previous methods of computing prime moduli transform the modulus operation into a more efficient mathematical form, but then provide no guidance to efficient hardware implementation. AMI is a general method that subsumes as special cases various mathematical tricks have been developed for special cases of binary division [26], [27]. AMI also has the advantage that both the **DIV** and **MOD** results are produced simultaneously from the same computation, which is important for efficient tag matching and 2-D cache mapping. We will first describe AMI and then use a detailed example to compare it with existing methods.

A. Efficient Index Implementation

The most efficient general solution to modulus computation to date uses the Chinese Remainder Theorem to convert **MOD**(N) to a narrow column of numbers which are then added modulo N. While certainly more efficient than a divide and remainder operation, there is implementation complexity in keeping the sum in **MOD**(N) terms. The performance advantage of our method is that it expresses both the **DIV** and **MOD** as a sum of narrow numbers in binary, which can be implemented using a high-performance array adder.

Instead of basing our derivation of **MOD** on modular arithmetic, we instead derive it from the original binary **DIV**/**MOD** operation as would occur with binary fixed-point numbers. AMI first multiplies the bits from the address by the reciprocal $(1/N)$ in fixed point. The **DIV** will be the integer part, while the **MOD** will be N times the fractional part of this result. This approach expresses the computation of **DIV**/**MOD** as two integer array multiplies, which on a 1 GHz GPU could complete in less than 2 cycles. Although this solution is already acceptable in terms of latency, it requires unnecessary area and power. From this logical baseline, we derive a far more efficient solution.

A sketch of the derivation of our index computation is as follows. First, we examine the binary form of the reciprocal of N, possibly expressing it as a difference of two numbers to reduce the number of ones. The number and spacing of ones ultimately determines the number and width of the additions needed to multiply by this reciprocal. Second, we derive a transformation to minimize the total number of additions and a hardware mechanism to compactly leverage sequences of infinitely repeating digits. Finally, we take

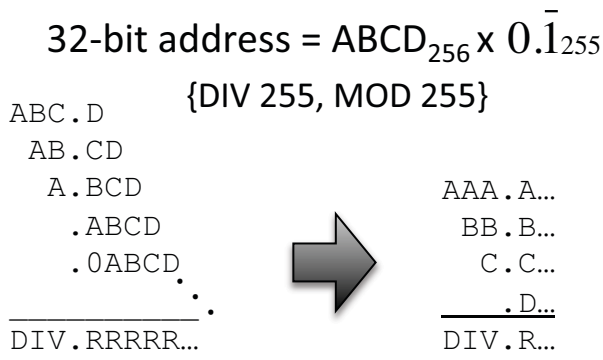


Figure 1: Key arithmetic transformation in AMI derivation.

the appropriate bits representing the resulting **DIV** and demonstrate how to trivially transform them into the **MOD** value. The end result can compute a **DIV** and **MOD** that in the best cases use fewer total binary adders than the number of bits in the source address.

We now explain the detailed algebraic derivation of our approach through a set of examples: First, we recreate the most efficient solutions known for computing **DIV**/**MOD** with moduli of form $2^N \pm 1$. Then, we show how AMI can generalize this to a single circuit computing **DIV**/**MOD** with moduli of the form $2^J \pm 2^K$. Finally, we show how AMI computes **DIV**/**MOD** efficiently for *any* modulus.

Derivation of Efficient $2^N - 1$ DIV/MOD: The goal of an index function that direct maps an address A to S banks or sets is to compute the address pair, $A \Rightarrow (A$ **DIV**(S), A **MOD**$(S))$. The modulus operator is the critical one, defining the bank or set. The floor divide operator is important in 2-D mappings (bank versus set) and as a tag option.

An efficient **MOD**$(2^N - 1)$ function is actually derived from an efficient truncated **DIV**$(2^N - 1)$ function. For generality of description, assume we will be computing in a logical base, $b = 2^N$, with each logical digit of the number containing N binary digits. We will begin computing $A/(2^N - 1)$ by multiplying A with the constant $1/(2^N - 1)$. This computation is only expressible in base b as an infinitely repeating decimal of the form $0.\overline{1}_b$. This multiplication may be viewed as adding an infinite number of A's together, each shifted over an additional N binary digits, or one logical digit in base $b = 2^N$. Figure 1 shows a concrete example of this process in which a 32-bit address A is mapped to 255 ($N = 8$) sets by expressing the source address as four base 256 digits, $abcd$. The result of the infinite sum is a three digit number in base 256, representing A **DIV**$(2^N - 1)$, and an infinite sequence of fractional digits R representing the remainder A **MOD**$(2^N - 1)$, again expressed in base 256.

To avoid an infinite number of addends, we apply associativity of addition to transform the sum such that each infinite diagonal represents one addend. Now, instead of adding an infinite number of finite digit numbers, we are adding a finite number (in this case just four) of infinite digit numbers. However, each number just repeats a single digit of A in base 256. As illustrated in Figure 1, we can now express the **DIV**/**MOD** operation concisely in this example of four base 256 digits ($A = abcd$) as the sum $(aaa.\overline{a} + bb.\overline{b} + c.\overline{c} + 0.\overline{d}) =$ **DIV**$(.\overline{R})$. We implement this computation efficiently by augmenting an array adder, where we capture the carry effects of an infinite number of digits by having a wrap-around carry at each stage where the binary carry-out of the remainder digit is fed into the carry-in of the next row. Since each digit is only $N = 8$ bits wide, the wires will be short.

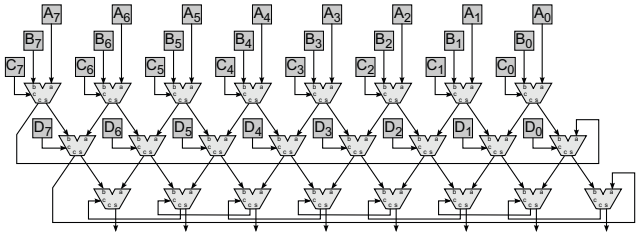


Figure 2: 8-bit wide augmented array adder implementation of **MOD**$(2^N$-1) in base 256.

The result of just the circular array addition is the N digit repeating pattern R_b in the infinite sequence of digits representing the **MOD** result $0.\overline{R}_b$. To convert this back to base $2^N - 1$, we just multiply $0.\overline{R}_b$ by $2^N - 1$. To do this quickly, we instead divide by $1/(2^N - 1)$, which we have already shown to be $0.\overline{1}_b$. Dividing $0.\overline{R}_b$ by $0.\overline{1}_b$ is simply R_b, our desired bank (or set) index. An alternative view is we are multiplying by 2^N and then subtracting $0.\overline{R}_b$, removing the repeating digits.

One side effect of preserving infinitely repeating digits (due to the identity $1.0 = 0.\overline{1}$) is that a result that should be in the form {**DIV**, **MOD**} = {**DIV**, 0} will instead map into {**DIV**-1, $2^N - 1$}. This is mathematically equivalent, but violates the base range. The standard mathematical answer can be achieved with a simple one-level circuit that checks the **MOD** value for all 1's and zeros it out, or by increasing the array width by one binary digit and setting the initial carry-in to 1. However, since we are only using the **MOD** value to choose a bank or set, we can omit even this circuitry and just map the SRAM banks as the high $2^N - 1$ values. This example highlights the difference between computing a mathematical **DIV**/**MOD** versus one equally suitable for removing bank conflicts.

Figure 2 shows such an array adder implementation using only 24 1-bit full adders to handle 32-bit addresses. Wider addresses need a few more logic levels to handle the extra digits. This operation can easily be folded into a base+offset computation within the same cycle. The rearrangement in Figure 1 further shows that we can compute A **DIV**$(2^N - 1)$ practically for free, as the intermediate sum as each digit is added represents each base b sum digit for the divide. This result matches the best previously reported implementation

of **MOD**$(2^N - 1)$ by Teng[25] and Dinechin[1].

Derivation of $(2^N + 1)$ **DIV/MOD:** As before, we start by multiplying A by $1/(2^N + 1)$, but reduce the cost by re-expressing the reciprocal in redundant binary notation $\{-1, 0, +1\}$. This constant is concisely represented in redundant form as $(0.\overline{10}_b - 0.\overline{01}_b) = 0.\overline{11'}_b$, where $1'$ denotes a digit with value -1. Expressing this multiplication and transforming the diagonals in the four digit example yields the sum $(AA'A.\overline{A'A}_b + BB'.\overline{BB'}_b + C.\overline{C'C}_b + 0.\overline{DD'}_b)$. This computation requires adding sums twice as wide as previously, because the length of repeating digits in the reciprocal is twice is wide. Viewing this sum as R_1R_{2b}, recovering **MOD** from the sum requires multiplying by $2^N + 1$, but this is simply $(R_1.R_2 + .R_1)_b$, which is trivial to fold into the array sum. This result matches the best implementation of **MOD**$(2^N + 1)$ by Dinechin[1].

Generalizing to $(2^J \pm 2^K)$ **DIV/MOD:** We extend the prior derivations by expressing **MOD**$(2^J \pm 2^K)$ as **MOD**$(2^N \pm 1) \times 2^M$. Standard modulo algebra can be re-expressed in binary as taking **MOD**$(2^N - 1)$ as before, then prepending the low order M bits from the **DIV** result. Since our approach already computes **DIV**, this would be a minimal extension, but we find an even simpler implementation. Take $(A \gg M)$ **MOD**$(2^N \pm 1)$, then append the low-order M bits from the original address A. Therefore, the circuit derived for **MOD**$(2^N \pm 1)$ also computes **MOD**$(2^J \pm 2^K)$, and we can scale any modulus by a factor of 2^N for free. Such **DIV/MOD** circuits resemble Figure 2 and require fewer total binary adders than bits in the address.

With minor additional multiplexing, we can modify the circuit computing $(2^J + 2^K)$ to also compute $(2^J - 2^K)$ by making the 1's complement optional and choosing just R_1 as the result. In general, wider AMI circuits can emulate narrower circuits with a small amount of configuration.

Generalizing to efficient DIV/MOD of ANY number: A slight generalization of our derivation methodology of **DIV/MOD**$(2^N + 1)$ produces an efficient circuit for any modulus M. First, find the repeating bit pattern of the reciprocal $1/M$, whose bit width depends on M. The address A is similarly divided into digits of this width. Then express the reciprocal redundantly. Starting from the right, every consecutive string of 3 or more 1s should be replaced by a difference, e.g. 0111 = 1001'. Multiple strings of 2 or more 1s separated by a single 0 can be expressed as a single large difference with the original zeros becoming -1s, e.g. 011011 = 1001'01'. Each 1 or -1 represents a single layer of logic (an array term), and this approach guarantees that total array terms remain less than half the number of bits in A, enabling 1-cycle completion for all moduli.

Our case study focuses on moduli between 33 and 61. In this range of 31 choices, 11 have 2 or fewer logic levels per digit, while more than half have 4 or fewer logic levels per digit and widths of 12 bits or less, while 25 have widths of 24 bits or less. Only four moduli, {37, 53, 59, 61} proved relatively inefficient. The total area of the array adder is proportional to the number of bits in the binary representation of A times the number of non-zero binary digits in the repeating reciprocal digit, shown in Figure 3.

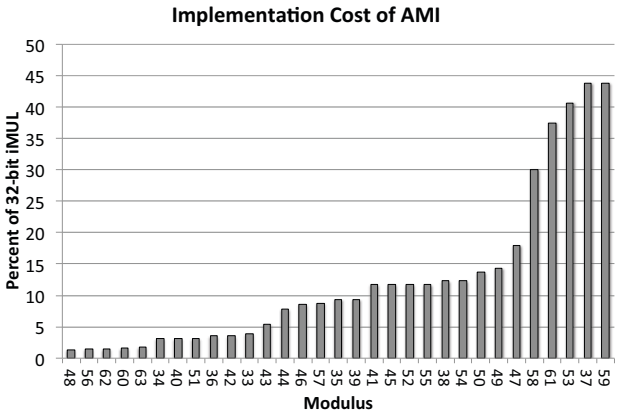


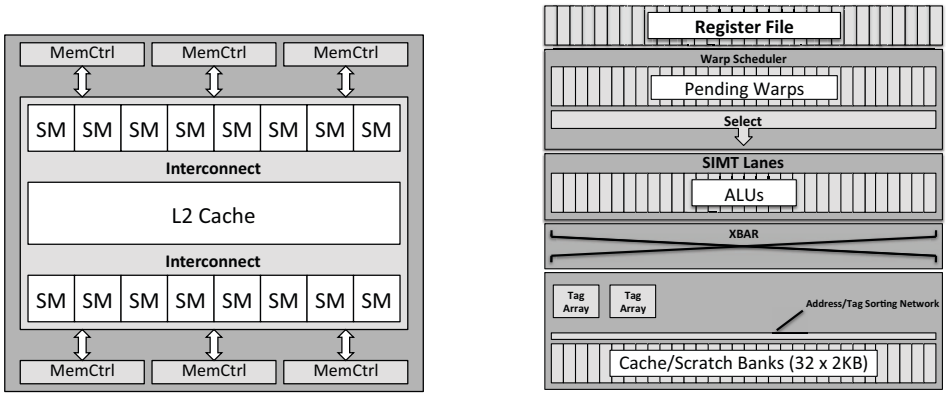
Figure 3: AMI Implementation cost versus MOD number.

When the repeating digit width is large, as in 4 of the 31 moduli above, a fallback approach is to dispense with the wrap around carry and use just enough digits to ensure that, with rounding, you will recreate exactly the correct **DIV** value from the reciprocal multiply. Error analysis indicates that we need to multiply A by a reciprocal truncated to one binary digit more than the width of A, setting the carry-in to 1 for rounding. Even this fallback approach is about 3x more efficient than an integer multiply and can still complete in one cycle. Nonetheless, using AMI gives us enough flexibility to choose more efficient moduli and still achieve full benefits. Two implementations used in this paper are **MOD**(62), an array addition of three 5-bit numbers and **MOD**(48), an array addition of 12 2-bit numbers, which can be expressed as short parallel sums to arbitrarily reduce gate delay.

B. Comparison With CRT

While efficient implementations have been derived for the special cases of **DIV/MOD**$(2^N \pm 1)$[1], [25], the only algorithm proposed for an arbitrary integer is based on the Chinese Remainder Theorem. We illustrate the competitive advantages of AMI with a concrete example. Sections VI and VII show that for global memory accesses, the best number of banks is 48. This number is also very amenable to interfacing with the rest of the memory system. The AMI derivation shows that since the reciprocal of 48 is $0.00\overline{01}$, the core circuit of the modulus computation of a 24-bit line address will consist of a 2-bit by 12-line array adder with wrap-around carry, with the last four bits of the modulus appended from the low-order bits of the address. This circuit is extremely compact and fast, and requires no special case analysis.

For an efficient implementation based on the Chinese Remainder Theorem (CRT) [25], we need to break the



(a) GPU chip

(b) Streaming multiprocessor

Figure 4: Baseline GPU architecture.

address into digits such that the coefficients are trivial to multiply. We choose the minimal digit size to hold the modulus, base 64. We note that 64^K **MOD**(48) is 16, so our coefficients shift the value left by 4-bits. We then compute the sum of 5 terms for each base 64 digit of the address as D_i **MOD**(48), sum these values, and then compute a final **MOD**. Because there is no circular carry, the sum requires extra bits, and the final **MOD** becomes non-trivial. At the very least, this procedure requires roughly 2 cycles to operate: one to compute the sum of terms, and one to take the **MOD** of that sum. However, we can only achieve 2 cycles if there is an efficient way to compute the individual **MOD**s of each addend with 48.

Even if a more elaborate analysis of residual arithmetic can separate the problem into a **MOD**(3) and **MOD**(16) problem, we would end up with a similar 2-bit $\times$ 12 array addition. However, the end sum would still require a full 6 bits as in the original case, so the array area will be closer to that required for a 6-bit $\times$ 12 array addition. Further, such a circuit still does not compute **DIV** as a byproduct.

Sophisticated CRT implementations (1) may require more elaborate (non-general) derivations to reformulate the problem to eliminate **MOD** operations on each term, (2) often result in larger array adders, and (3) in general require an extra cycle to compute the final **MOD**. AMI outperforms even the best known implementations using CRT.

IV. Throughput Architecture Background

In the remainder of the paper we present a case study analyzing the effectiveness of our AMI technique in reducing bank and set conflicts in a modern GPU. Our baseline architecture is loosely modeled on NVIDIA's Fermi architecture, shown in Figure 4. This is a generic design similar to others in the literature [28], [29], [30], and is not intended to correspond directly to any existing product. The GPU consists of 16 *streaming multiprocessors* (SMs), each containing 64 SIMT (single-instruction, multiple thread) lanes that each execute up to one thread instruction per cycle. Each SM dynamically manages up to 8 different CTAs (Cooperative Thread Arrays) simultaneously, populating up to 48 warp slots with up to 1,536 total active SIMT lanes. Up to two 32-lane warp instructions per cycle issue in order and complete out of order. Each warp controller contains a small instruction buffer and scoreboard to prepare instructions for issue, and a reorder buffer for in-order retirement.

Each SM contains a 128KB SRAM array for storing register values and a 64KB SRAM scratchpad, half of which is used as a 32KB L1 cache. The baseline SRAM array is divided into 32 fully independent 32-bit wide banks and supports full scatter/gather via a bank sort network and a 32$\times$32 32-bit crossbar. Global tag arrays determine if an L1 cache line is present. Each L1 cache line is 128-bytes, spanning 32 banks. Our L1 cache is direct mapped. While slightly pessimistic, GPUs have limited associativity to conserve power, and AMI provides an alternate approach to reducing set conflicts. L1 misses coalesce in MSHRs and are sent to a 768KB global L2 cache. L2 misses go directly to DRAM. As all our benchmark kernels fit in a few KB, we ignore instruction cache misses.

Active lanes in the warp may not all successfully complete their memory transaction due to (1) a conflict in the tag array (too many cache lines referenced), (2) memory divergence (not all cache lines present in L1), or (3) bank conflicts. Incomplete memory instructions move to a replay buffer which contends with new instructions for issue slots. To preserve memory consistency, other in-flight memory instructions from the same warp are squashed and reissued. Our model is more aggressive than current GPUs, which avoid a replay queue by squashing and reissuing all instructions from the warp beginning with the conflicted memory instruction [31], [32]. Replays reuse major SM hardware and preserve memory consistency but also cost energy and reduce throughput performance [33].

To demonstrate a meaningful improvement over a state-of-the-art architecture, we chose an aggressive base case with dual global tag arrays that virtually eliminates tag conflicts while increasing average base performance by 10%. We examine techniques based on AMI for reducing bank

Table I: Benchmarks.

Rodinia Benchmarks		Characteristic
backprop	Back Propagation	unstructured grid
needle	Needleman-Wunch	dynamic programming
hotspot	physics simulation	structured grid
srad	image processing	structured grid
lu	LU Decomposition	dense linear algebra
hwt	Heart Wall	structured grid
Parboil Benchmarks		**Characteristic**
sad	sum of absolute differences	structured grid
cp	distance cutoff coulombic potential	unstructured grid
tpacf	two point angular correlation function	structured grid
mri-q	scanner calibration	structured grid
mri-fhd	compute 3D image	unstructured grid
rpes	molecular dynamics simulation	graph processing
pns	petri net simulation	graph traversal

conflicts during scratchpad access and bank and set conflicts during L1 cache access. In Figure 4, one AMI unit is placed in each address generation unit (one per lane for a total of 32) and one is placed at each cache tag array when applying AMI to sets.

V. Methodology

Simulator: We created a custom GPU architecture simulator that models long latency replay mechanisms, sort networks, crossbars, and a GPU-like primary memory system. Our simulator is a fully execution-driven, highly detailed, cycle-accurate simulation of the entire GPU core and secondary memory system. We fully model the execution pipelines, thread selection and retirement, multiple pipelines in the primary memory system, tag and bank access, reorder buffers, sort networks and crossbars, wire traversals, and contention at all levels. Our performance numbers are the total cycles taken by each application as simulated; all benchmarks are simulated to completion.

For efficiency, we leverage a modified version of Ocelot [34], [35] for the interpretation and functional execution of instructions in CUDA programs. Ocelot then feeds a cycle-accurate timing simulation that is able to model the conflicts described in Section IV fast enough to run the largest possible datasets on all benchmarks for billions of cycles, realistically stressing the memory system.

For this paper, we ran benchmarks on a single SM and allocated them a single, direct mapped, 48KB slice of the 768KB L2 cache. This simplification results in somewhat conservative L2 cache performance, since SMs cannot benefit from shared data. However, unlike the findings in [36], we found that global data sharing across tasks (CTAs) is minimal in the applications we evaluated.

Power model: Modeling power requirements is challenging, as the structures examined in this paper are not modeled in conventional simulators [37], and the area and power required depend highly on the exact VLSI implementation. For replay costs, we estimated core pipeline power and redundant register reads as a fixed fraction of SM power in the manner of [38]. Power in the primary memory pipeline is dominated by the cross-bar switch, whose power is best approximated by a wire-transmission model. We modeled the area and power of the cross-bar similar to CACTI [39], using a standard cell library for geometry and computing active power based on actual wire traversals during simulation. We chose the conservative VLSI layout of a monolithic grid connecting data lanes to SRAM banks, and modeled active and static power using ITRS Roadmap wire models, taking the approach of [40]. The power needed for register accesses, tag accesses, SRAM banks, and L2 caches, as well as the size and power for crossbar logic came from standard cell data reported in [41]. Power and size of AMI circuits were conservatively scaled from 3-bit full-adder results from the same paper.

Benchmarks: Table I shows the Rodinia [42] and Parboil [43] benchmarks used in our study. We always use the largest standard input data sets available for the benchmarks to stress the memory system in the most realistic manner possible, with typical benchmarks running for tens of billions of cycles. For our characterization, we also count memory accesses in an architecturally relevant way to capture the behavior of (1) coalescing of scalar thread memory accesses into cache lines, and (2) coalesced scalar references from multiple threads in a warp to the same address.

VI. Benchmarks and Moduli

This section characterizes the memory access behavior of our benchmarks and the sensitivity of this behavior to different numbers of L1 and scratchpad banks.

A. Memory Access Patterns

Stereotypical views of throughput applications include extremely high arithmetic intensity (low memory intensity), little to no reuse of data (streaming applications), and regular, sequential data access, which would make notions of bank conflicts irrelevant. However, we find that the majority of these benchmarks exhibit more intense and more complex memory behavior.

Figure 5 shows how values are reused across threads and warps for benchmarks with more than 0.01% *memory intensity*, defined as the fraction of instructions that are `load` or `store` operations. Some benchmarks have two distinct phases, labeled A and B. We found most inter-task (CTA) sharing of data was among small groups of tasks separated by large distances in space and time, requiring complex task scheduling to leverage.

In the figure, a *vector broadcast* represents a CTA-wide access to the same value that need only be fetched from the memory system once. A *warp broadcast* represents the same concept, but limited to the extent of a warp (32 threads). A *lane broadcast* represents the same concept but for a partial warp. Finally, *scatter/gather* represents individual thread accesses to different addresses. Roughly half the benchmarks

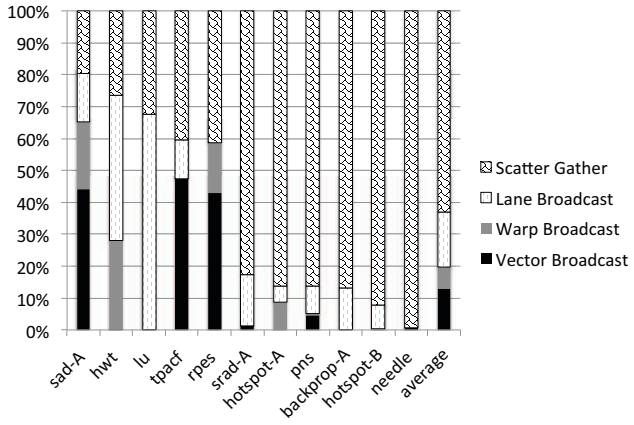


Figure 5: Categorization of GPU memory accesses.

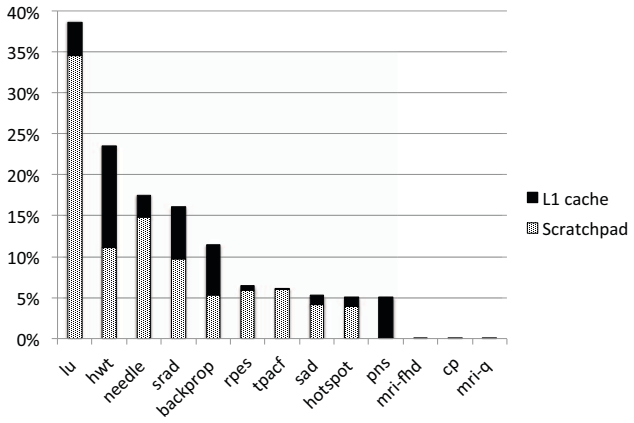


Figure 6: Memory Intensity

have more than 50% of all memory instructions as some form of scalar broadcast, which results from an architecture that has a vector nature, but lacks scalar registers to span vector elements. As the architecture already coalesces these accesses to eliminate unnecessary bank conflicts, we do not consider them in our bank conflict analysis. A second observation is that a significant fraction of accesses cannot be completely coalesced (as emphasized by lane broadcasts and part of the scatter/gathers), subjecting them to both tag and bank conflicts.

B. Memory Conflicts

Bank conflicts tend to be more common and expensive in GPUs than in conventional architectures. In current systems, each bank conflict triggers one or more instruction replays, which costs the processor throughput (issue) slots, lowers total throughput performance, and uses extra power. Replaying memory access instructions multiple times adds large amounts of latency that is difficult for threads and available parallelism to cover. The impact of bank conflicts depends both on memory access intensity and on the bank access pattern.

Memory intensity: Figure 6 shows the memory intensity of the benchmarks, which is critical in interpreting

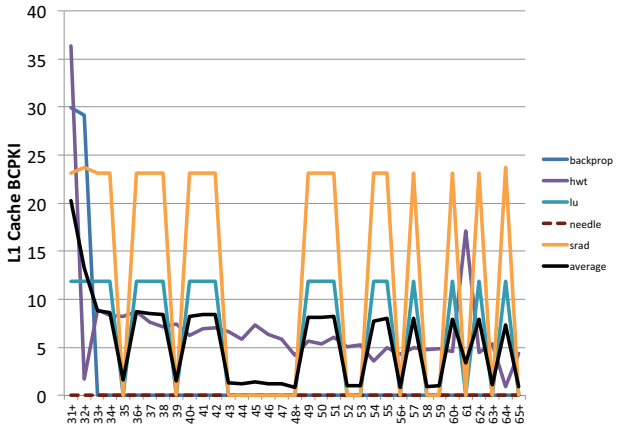


Figure 7: Benchmark sensitivity to L1 bank count.

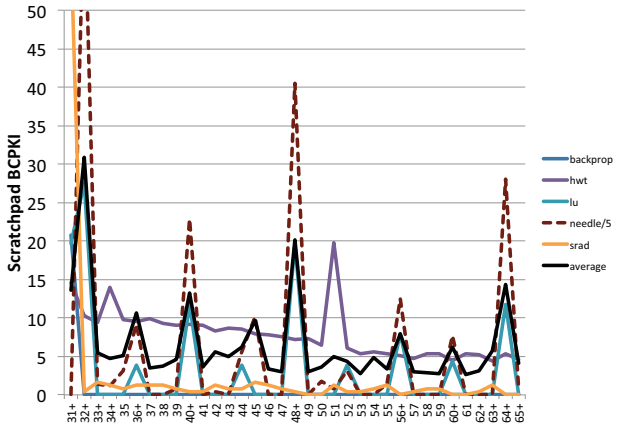


Figure 8: Benchmark sensitivity to scratchpad bank count.

the severity of conflicts, as conflicts effectively multiply memory intensity. Five benchmarks, lu, hwt, needle, srad and backprop, have at least 10% memory intensity, which we will group together as *high memory intensive* benchmarks. The next 5 benchmarks have roughly 5% memory intensity, rpes, tpacf, sad, hotspot, and pns, which we classify as *low memory intensive* benchmarks, while the last 3 benchmarks, mri-fhd, mri-q and cp, read from constant memory and apply reductions, so they have essentially no memory intensity. The chart also subdivides the memory references into L1 cache accesses and scratchpad accesses. L1 cache accesses are subject to both set conflicts and bank conflicts, while scratchpad accesses are only subject to bank conflicts.

We focus primarily on the five high memory intensive benchmarks since those will show the greatest effects from changes in the memory system. We found that optimizations to the memory system do not degrade performance for the benchmarks with low or no memory intensity (Section VII-B).

C. Sensitivity to Number of Banks

Figures 7 and 8 show the sensitivity of bank conflicts to bank count for L1 cache and scratchpad accesses, assuming constant L1 or scratchpad capacity, respectively. In contrast to the ideal results above, these and all subsequent results in the paper are produced using a detailed cycle accurate simulation of the memory system. In each graph, the y-axis is the number of bank conflicts per 1000 instructions (BCPKI). The x-axis sweeps the bank count and indicates with a "+" those moduli which are particularly easy to compute, as described in Section III. In general, the number of bank conflicts decreases with increasing bank count. However, the figures show significant spikes with scratchpad bank counts at multiples of 8, indicating systematic conflicts between bank count and the memory access stride. The benchmark hwt has irregular memory access patterns, leading to fewer conflicts than the other more strided applications. The L1 cache is less sensitive to bank count and its bank conflicts often spike at different bank counts than the scratchpad.

These figures show that the prime numbers are not necessarily the best choice for bank counts. For the L1 cache, the bank conflicts fall to zero at a bank count of 48, which not only is not prime but also is not an odd number, and many even numbers are good candidates for both L1 and scratchpad.

VII. Performance Results

As indicated above, this section uses a detailed, cycle-accurate simulation to evaluate the performance of AMI in the context of a full secondary memory system.

Bank and moduli configurations: AMI indexing can be disabled or configured to index by any modulus choice of sets or banks *less than* the existing physical sets or banks. For sets, using less than the total number of cache lines is a net benefit. For banking, in the context of a GPU, we cannot constructively index less than 32 banks without (in most cases) increasing bank conflicts, so altering the physical scratchpad/L1 is necessary.

We evaluate two different physical scratchpad augmentations. In one case, we simply add a 33rd or 34th SRAM bank, increasing area and leakage power by 3-6%, but having minimal impact on the crossbar networks. This enables indexing modulo 32 to 34 within the scratchpad or L1 cache. The L1 and scratchpad are disjoint, so they can freely use different index moduli. The second approach is to keep the cache size the same, but convert it to 64 half-sized banks, doubling the crossbar size from 32:32 to 32:64. The cost of this change is negligible in power and area (Section VII-D), and it allows indexing with any moduli up to 64 for both the L1 cache and scratchpad. For the L1 cache, cache lines are always maintained as 128 consecutive bytes, regardless of the number of banks, to preserve compatibility with the rest of the memory system. In the following sections, the term "bank count" may be used interchangeably with the indexing modulus and does not represent the underlying physical bank count.

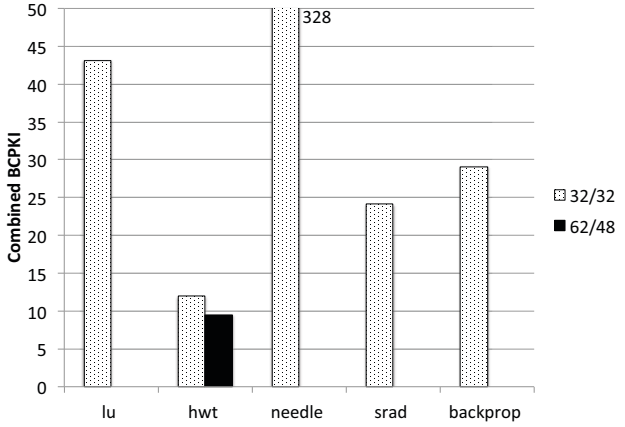


Figure 9: Total bank conflicts per thousand instructions for baseline vs AMI-Banking.

While we present performance sweeps in Section VII-C, in the next section we highlight a few sample moduli to illustrate the benefits of AMI at different physical bank configurations. We chose 34 and 62 as two extremes for scratchpad access, and 48 in the middle for L1 cache access. None of these moduli have been studied before, and all are easy to compute (Figure 3) and have robust performance across all benchmarks, achieving near optimum in both performance and power.

A. AMI Bank Conflict Reduction

As shown in Figure 8, for the base case of 32 banks, four of the five memory intensive benchmarks have severe bank conflict issues of 20% or greater. One benchmark, hwt, only has moderate conflicts of 12%, due to irregular memory access patterns, which tend to be less pathological. Figure 9 demonstrates that applying AMI-banking with a single, static number of banks across all benchmarks and kernels reduces 98% of bank conflicts. In the figure, the gray bar represents the baseline architecture with 32 banks for the scratchpad and 32 banks for the L1 cache. The AMI enhanced architecture uses 62 banks for the scratchpad and 48 banks for the L1 cache. In 4 of the 5 benchmarks with the most severe conflict issues (backprop, lu, needle, and srad), conflicts are reduced to zero, without requiring any programmer intervention. The one outlier, hwt, has an irregular memory access pattern, making it less amenable to AMI improvements, but also resulting in fewer initial conflicts.

B. AMI Performance

Speedup: Figure 10 shows the increase in throughput performance resulting from bank and set conflict reduction. Our notation in the figures is (#SP banks/#L1 banks), with S+ denoting the addition of AMI-sets. Applying AMI to scratchpad banks and using the cheapest solution of 33 banks

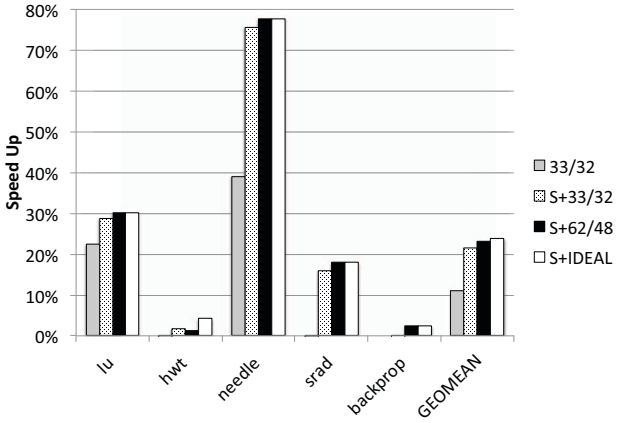


Figure 10: AMI speedup over base case of 32/32 banks and 2 tag arrays.

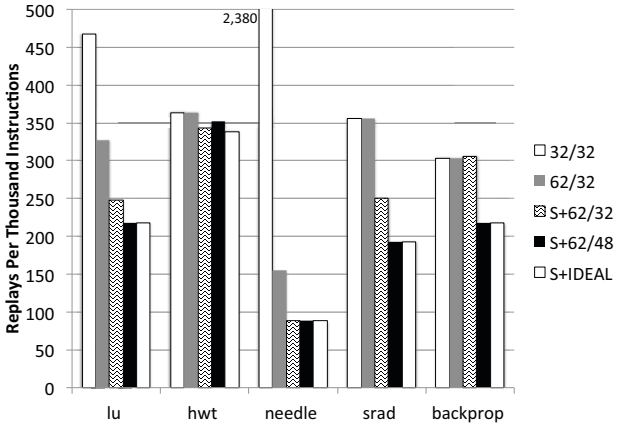


Figure 11: Reduction in replays from applying AMI.

yields a geometric mean 11% speedup, or nearly half the ideal speedup for no bank conflicts. Applying AMI-sets to the L1 cache, an extremely low cost optimization, increases aggregate speedup to 21.4%. For execution speed, supporting the two lowest cost AMI implementations (S+33/32) already achieves 90% of ideal speedup. Finally, applying a more aggressive choice of static AMI bank numbers to both the scratchpad and L1 cache leads to a 23.1% geometric mean speedup, or 97% of ideal speedup. This performance from static bank counts suggests that dynamic AMI mappings per benchmark are not likely to be worth the additional complexity, at least for this workload.

Three benchmarks (lu, needle, and srad) see large throughput performance gains of 18%–78%, while hwt and backprop see moderate performance gains of 2.4–4.2%. Irregular memory access patterns and a high ratio of load-dependent instructions prevent hwt from seeing a large increase. The limited effects in backprop are due to cache misses that mask the additional latency caused by local replays. These overall improvements are very significant for a throughput architecture, which by its very nature is designed to mask high latency through massive task level parallelism, and they are achieved over a very aggressive base case, with very limited application of AMI.

Finally, we note that there were no deleterious effects for the 8 remaining low memory intensity benchmarks. Those benchmarks showed a geometric mean speedup of 0.94%, the largest winner being hotspot, with a 4.92% speedup, and sad and tpacf, with 1.66% and 1.33% speedups, respectively. Only one benchmark, pns, saw a negligible (0.06%) slow down, which can be avoided by switching off AMI.

Replay and Energy Reduction: Figure 11 shows total replays per thousand instructions (RPKI). Comparing with Figure 9, we see that bank conflicts can increase replays by an order of magnitude since the relatively long pipelines of throughput architectures lead to a large number of instructions in flight.

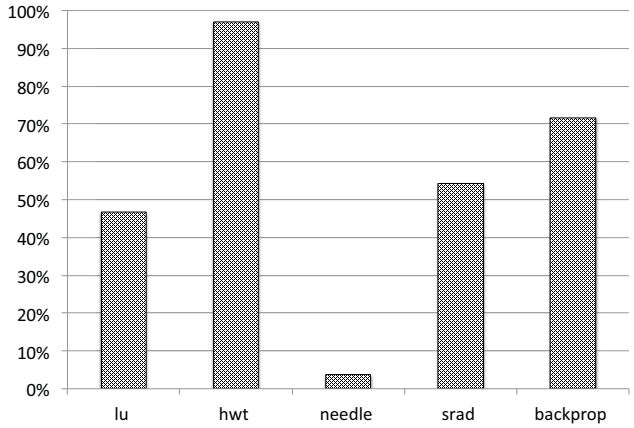


Figure 12: Fraction of total instructions executed relative to baseline 32/32.

Overall, 4 of the 5 memory intensive benchmarks show absolute reduction in replays of nearly 20% or more, while 3 have replays reduced by 50% or more. Hwt only showed a modest reduction in replays of about 7%, owing both to its limited reduction in bank conflicts from AMI due to its irregular memory access pattern and the low number of instructions associated with each replay. Two of the benchmarks had substantial replay reductions from just applying AMI to scratchpad banks and four of the benchmarks benefited from applying AMI to L1 cache sets. Finally, 3 of the benchmarks had significant improvements from applying AMI to L1 banks, despite L1 banks having minimal impact on execution speed. All benchmarks were able to achieve nearly ideal (conflict-free) levels of replay reduction with appropriate use of AMI.

Figure 12 shows the fraction of total instructions executed by (S+62/48) relative to the baseline 32/32 architecture. The reduction in executed instructions stems directly from the set and bank conflict replays eliminated by AMI. The benchmark needle sees the most impressive improvement, reducing its executed instruction count by about 95%. In-

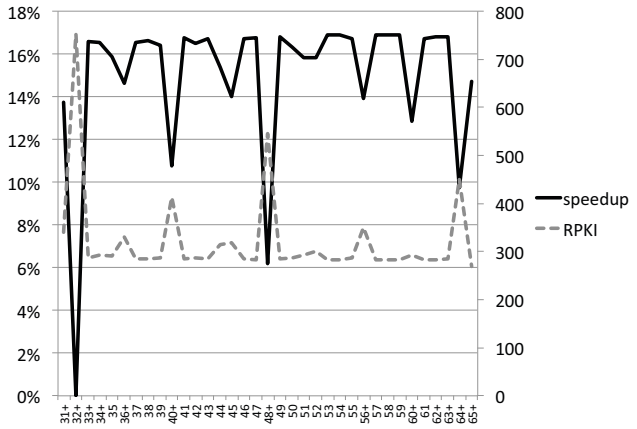


Figure 13: Speedup and RPKI vs number of scratchpad banks.

structions executed can be used as a first order proxy for power consumption, indicating that any power costs for AMI are offset by the benefits from reduced replays.

C. Performance sensitivity to bank count

Figure 13 shows the variation in speedup and replays when varying scratchpad bank count and holding the rest of the system constant. Bank counts with a + represent those with the simplest AMI index computation. The graph shows significant performance benefits (up to 17% speedup or 62% replay reduction) from just altering the number of scratchpad banks. About a dozen different bank counts are comparable in achieving most of the performance benefits in both speedup and RPKI. This effect is important, first, because it only requires a small number of indexing options to optimize all benchmarks, so a single, static choice can achieve near ideal performance over all benchmarks. Second, ease of index computation is not the only integration consideration in a system, and certain moduli may be favored for architectural reasons.

While all benchmarks benefit from having a bank modulus other than 32, there are variations in performance with moduli, and different benchmarks (and kernels) see better results at different moduli. Furthermore, optimal moduli differ between scratchpad and L1-cache, and differ for throughput performance vs energy savings (L1 sensitivities in Figure 7).

AMI allows hardware designers to make clean tradeoffs between index cost, networking cost, average speedup, average energy savings, and robustness across benchmarks, at a relatively insignificant cost in power, area, or complexity.

D. Integration costs

AMI circuits: We need to add 32 AMI circuits per core (one for each ALU lane), and two for the tag arrays. Active and leakage power are highly dependent on the exact VLSI implementation of a circuit used. As a conservative estimate of power, we extrapolate power and area figures from standard cells used in [41], which were based on 32nm BSIM-4 predictive technology models (PTM). The AMI circuit is very similar to a ripple adder with the same number of bits as the address in the simplest case, or double that in the next simplest case. Linear extrapolation of the 3-bit adder cell from [41] leads in the worse case to 222fJ energy per DIV/MOD computation and an area of 8.6um^2, representing overheads of 0.5% in active power and 0.1% in leakage power compared to the 32-bit FMAD in each lane. Performing an AMI index computation for each lane for every global and scratchpad memory access results in an average addition of 0.68 milliwatts active power, ranging from 0 to 2.26 milliwatts for the highest benchmark intensity. The delay of AMI circuits is less than one cycle, but even if AMI forced adding a cycle to the pipeline, it would have no visible impact on performance, because GPU scratchpad and cache latency is tens of cycles and there is ample time to run AMI in parallel, off the critical path.

L1 sets: A single AMI circuit is used at each tag array to restrict tag lookup to less than the total amount. The cost of an AMI circuit is negligible in power, area, and delay, and the reduction in set conflicts more than offsets having fewer cache lines. However, because a set location now depends on more of the address, additional tag bits are needed, the amount depending on the modulus. In the worse case, storing an extra 8 bits per tag entry only increases scratchpad area by half a percent. Our implementation retains the notion of 128-byte cache lines, requiring no changes to the TLB or L2 cache interface. Mapping a given cache line to a physical location involves rotation, which in our case is provided by the existing crossbar network.

Scratchpad/L1 banks: To implement true scatter gather, a large crossbar connects 32 ALU lanes to the default 32 scratchpad/L1 banks. Implementing AMI involves changing the crossbar to 32×N, and possibly adding additional SRAM banks. For a choice of 33, adding one more SRAM bank would be the most sensible, while a choice of 62 would likely double the bank count and crossbar size. When adding banks to the L1/scratchpad, active access power is actually reduced, because the same number of bank accesses occur on smaller banks. Leakage power increases slightly due to the overhead of control circuitry as banks become smaller. Adding a single bank only reflects a 3% increase in area, while doubling the number of banks also only requires a minimal increase in area and leakage power due to increased overhead per bank.

Crossbar: The largest change in the crossbar network moves from 32×32 to 32×64, doubling the number of access muxes and wire links. We simulated link by link crossbar traversals for all benchmarks. Our simulation modeled a monolithic rectangular crossbar in which each link activation was measured. The average data traversal length was a little less than the average manhattan distance due

to lane zero being used with greater frequency. We then conservatively estimated total power as roughly double the cost of simulated wire energy of that configuration to account for active logic energy. Linearly estimating the area of the SRAM banks from [41] and using wire capacitance and minimum spacing for lanes from the ITRS Roadmap [44], the active wire power at 64 banks ranges from 0.6 milliwatts to 1.9 milliwatts. Doubling that as a total estimate yields 1.2 milliwatts, which varies linearly with the number of banks chosen. In comparison, the energy to access the SRAM banks is an order of magnitude greater, at 116 milliwatts, and access to registers in the core pipeline is even higher. Thus the extra power needed for AMI banking is a small fraction of core power.

VIII. Conclusion

In this paper, we introduce a new scheme for address mapping using Arbitrary Modulus Indexing (AMI). We show that for non-power-of-2 indexing, our scheme can be implemented as or more efficiently than previous schemes, even for the most competitive case of Mersenne prime indexing. We then show that set and bank conflicts in the primary memory system of a replay-style throughput architecture have a deleterious effect on system performance caused by the alignment of the bank count with natural strided memory access patterns across the threads in a warp, similar to previously known effects for vector supercomputers. We show how to use AMI to obtain significant gains in performance and power efficiency on such an architecture.

The resulting system is simple to implement and provides robust benefits across all of our benchmarks, on average eliminating 98% of bank conflicts, and completely eliminating bank conflicts on 4 of the 5 benchmarks with the most serious conflict issues, with no performance detriments for benchmarks with low memory intensity. Applying AMI to scratchpad banks, L1 banks, and L1 sets achieved 98% of ideal performance, resulting in a geometric mean speedup of 24% across the 5 most memory intensive benchmarks, significant for an aggressive baseline architecture designed to mask latency with explicit TLP. AMI also resulted in a 64% reduction in instruction replays. The cost of our implementation is just a few percent of the area and active power of a 32-bit array multiplier and adds just a few gate delays to the pipeline, while worst case power increases due to sort networks and cross bars is under 2 milliwatts.

AMI provides great design flexibility, enabling trade-offs in throughput performance, power reduction, and implementation cost. Our scheme works for all moduli, and relatively efficient implementations of our scheme apply to a large number of moduli. This opens up the possibility of separating the indexing modulus from the physical implementation of a cache or memory banks and tuning the modulus for best performance on a given workload. In fact, it is feasible to implement a single mapping circuit that can be dynamically tailored to moduli best suited for individual applications, although our performance results indicate this is unnecessary for the benchmarks we used. AMI benefits are orthogonal to, and can enhance other indexing schemes designed to work on power of 2 sized caches. Given this flexibility, we expect schemes based on AMI to be useful for other aspects of the memory systems of various types of architectures.

Acknowledgment

The authors would like to thank the numerous anonymous reviewers for their helpful feedback and Benoît D. de Dinechin for providing valuable insights into prior work. This work was supported in part by the National Science Foundation under award CCF-0916745. Simulation work was significantly enhanced by servers donated by Intel under their Academic Support Program.

References

[1] B. D. de Dinechin, "A Ultra Fast Euclidean Division Algorithm for Prime Memory Systems," in *Proceedings of the ACM/IEEE Conference on Supercomputing*, November 1991, pp. 56–65.

[2] Q. Yang and L. W. Yang, "A Novel Cache Design for Vector Processing," in *Proceedings of the International Symposium on Computer Architecture*, May 1992, pp. 362–371.

[3] A. Seznec and J. Lenfant, "Odd Memory Systems May be Quite Interesting," in *Proceedings of the International Symposium on Computer Architecture*, May 1993, pp. 341–350.

[4] T. Sun and Q. Yang, "A Comparative Analysis of Cache Designs for Vector Processing," *IEEE Transactions on Computers*, vol. 48, no. 3, pp. 331–344, March 1999.

[5] S. Che, J. W. Sheaffer, and K. Skadron, "Dymaxion: Optimizing Memory Access Patterns for Heterogeneous Systems," in *International Conference on High Performance Networking and Computing (Supercomputing)*, November 2011.

[6] B. L. Chamberlain, S. J. Deitz, D. Iten, and S.-E. Choi, "User-defined Distributions and Layouts in Chapel: Philosophy and Framework," in *Proceedings of the USENIX Conference on Hot Topics in Parallelism*, May 2010.

[7] T. Chilimbi, M. Hill, and J. Larus, "Making Pointer-based Data Structures Cache Conscious," *IEEE Computer*, vol. 33, no. 12, pp. 67–74, December 2000.

[8] B. Bershad, D. Lee, T. Romer, and J. Chen, "Avoiding Conflict Misses Dynamically in Large Direct-mapped Caches," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems*, October 1994, pp. 158–170.

[9] A. Seznec, "A Case for Two-way Skewed-associative Caches," in *Proceedings of the International Symposium on Computer Architecture*, May 1993, pp. 169–178.

[10] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee, "Using Prime numbers for Cache Indexing to Eliminate Conflict Misses," in *Proceedings of the International Symposium on High-Performance Computer Architecture*, February 2004, pp. 288–299.

[11] M. Kharbutli, Y. Solihin, and J. Lee, "Eliminating Conflict Misses using Prime Number-based Cache Indexing," *IEEE Transactions on Computers*, vol. 54, no. 5, pp. 573–586, May 2005.

[12] T. Givargis, "Improved Indexing for Cache Miss Reduction in Embedded Systems," in *Design Automation Conference*, June 2003, pp. 875–880.

[13] K. Patel, E. Macii, L. Benini, and M. Poncino, "Reducing Cache Misses by Application-specific Re-configurable Indexing," in *International Conference on Computer Aided Design (ICCAD)*, November 2004, pp. 125–130.

[14] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina, and A. Seznec, "Tarantula: a Vector Extension to the Alpha Architecture," in *Proceedings of the International Symposium on Computer Architecture*, May 2002, pp. 281–292.

[15] M. Valero, T. Lang, and E. Ayguadé, "Conflict-free Access of Vectors with Power-of-two Strides," in *International Conference on High Performance Networking and Computing (Supercomputing)*, November 1992, pp. 149–156.

[16] B. R. Rau, "Pseudo-randomly Interleaved Memory," in *ACM SIGARCH Computer Architecture News*, vol. 19, no. 3, May 1991, pp. 74–83.

[17] A. Agarwal and S. D. Pudar, "Column-associative Caches: A Technique for Reducing the Miss Rate of Direct-mapped Caches," in *Proceedings of the International Symposium on Computer Architecture*, May 1993, pp. 179–190.

[18] C. Zhang, X. Zhang, and Y. Yan, "Two Fast and High-associativity Cache Schemes," *IEEE Micro*, vol. 17, no. 5, pp. 40–49, Sept./Oct. 1997.

[19] D. Sanchez and C. Kozyrakis, "The ZCache: Decoupling Ways and Associativity," in *Proceedings of the International Symposium on Microarchitecture*, December 2010, pp. 187–198.

[20] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi, "Cuckoo Directory: A Scalable Directory for Many-core Systems," in *Proceedings of the International Symposium on High-Performance Computer Architecture*, February 2011, pp. 169–180.

[21] E. Hallnor and S. Reinhardt, "A Fully Associative Software-managed Cache Design," in *Proceedings of the International Symposium on Computer Architecture*, May 2000, pp. 107–116.

[22] D. Lawrie and C. Vora, "The Prime Memory System for Array Access," *IEEE Transactions on Computers*, vol. 100, no. 5, pp. 435–442, May 1982.

[23] T. Austin, D. Pnevmatikatos, and G. Sohi, "Streamlining Data Cache Access with Fast Address Calculation," in *Proceedings of the International Symposium on Computer Architecture*, June 1995, pp. 369–380.

[24] Q. S. Gao, "The Chinese Remainder Theorem and the Prime Memory System," in *Proceedings of the International Symposium on Computer Architecture*, May 1993, pp. 337–340.

[25] M. Teng, "Comments on "The Prime Memory Systems for Array Access"," *IEEE Transactions on Computers*, vol. 100, no. 11, pp. 1072–1072, November 1983.

[26] M. Calhoun, "On the Binary Decimal Expansion of the Reciprocal Prime's," 2010. [Online]. Available: http://math.stackexchange.com/questions/3976/on-the-binary-decimal-expansion-of-the-reciprocal-primes

[27] H. S. Warren, "Hacker's Delight, 2nd edition." [Online]. Available: http://www.hackersdelight.org/divcMore.pdf

[28] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, April 2009, pp. 163–174.

[29] NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," http://nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2009.

[30] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU Microarchitecture through Microbenchmarking," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, March 2010, pp. 235–246.

[31] P. Micikevicius, "GPU Performance Analysis and Optimization," *GPU Technology Conference*, http://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/S0514-GTC2012-GPU-Performance-Analysis.pdf, May 2012.

[32] A. L. Minkin, S. J. Heinrich, R. Selvanesan, C. McCarver, S. G. Carlton, M. Y. Siu, Y. Y. Tang, and R. J. Stoll, "Cache Miss Processing Using a Defer/Replay Mechanism," USA Patent 8,266,383 B1, September 11, 2012.

[33] A. E. Turner, "On Replay and Hazards in Graphics Processor Units," *UBC Masters thesis*, https://circle.ubc.ca/handle/2429/43493, June 2012.

[34] A. Kerr, G. Diamos, and S. Yalamanchili, "GPUOcelot - A Binary Translator Framework for PTX," October 2009. [Online]. Available: http://code.google.com/p/gpuocelot

[35] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark, "Ocelot: A Dynamic Compiler for Bulk-Synchronous Applications in Heterogeneous Systems," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, September 2010, pp. 353–364.

[36] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Orchestrated Scheduling and Prefetching for GPGPUs," in *Proceedings of the International Symposium on Computer Architecture*, June 2013, pp. 332–343.

[37] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. Aamodt, and V. Reddi, "GPUWattch: Enabling Energy Optimizations in GPGPUs," in *Proceedings of the International Symposium on Computer Architecture*, June 2013, pp. 487–498.

[38] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-efficient Mechanisms for Managing Thread Context in Throughput Processors," in *Proceedings of the International Symposium on Computer Architecture*, June 2011, pp. 235–246.

[39] P. Shivakumar and N. P. Jouppi, "Cacti 3.0: An Integrated Cache Timing, Power, and Area Model," Technical Report 2001/2, Compaq Computer Corporation, Tech. Rep., 2001.

[40] P. Kogge *et al.*, "ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems," University of Notre Dame, Tech. Rep. TR-2008-13, 2008.

[41] X. Guo, E. Ipek, and T. Soyata, "Resistive Computation: Avoiding the Power Wall with Low-leakage, STT-MRAM Based Computing," in *Proceedings of the International Symposium on Computer Architecture*, June 2010, pp. 371–382.

[42] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *International Symposium on Workload Characterization*, October 2009, pp. 44–54.

[43] "Parboil Benchmark Suite," http://impact.crhc.illinois.edu/parboil.php. [Online]. Available: http://impact.crhc.illinois.edu/parboil.php

[44] "International Technology Roadmap for Semiconductors (ITRS), 2011 Edition." [Online]. Available: http://www.itrs.net/Links/2011ITRS/Home2011.htm